AD-A085 516 VIRGINIA POLYTECHNIC INST AND STATE UNIV WASHINGTON --ETC F/G 9/2
DIALOG-A SIMULA CLASS FOR WRITING INTERACTIVE PROGRAMS.(U)
OCT 79 R J ORBASS, R E PORTER AFOSR-70-0021
UNCLASSIFIED VPI/SU-TN-79-3A AFOSR-TR-80-0446 NL

1 OF 1
AD A085 516

END
DATE FILMED
7-80
DTIC

EXTENSION DIVISION

VIRGINIA POLYTECHNIC INSTITUTE AND STATE UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE
GRADUATE PROGRAM IN NORTHERN VIRGINIA

P. O. Box 17186
Washington, D. C. 20041
(703) 471-4600

DIALOG-

A SIMULA* CLASS FOR WRITING INTERACTIVE PROGRAMS†§

Richard J. Orgass

and

Robert E. Porter

Technical Memorandum No. 79-3a
October 14, 1979

VPI/SU-TM

ABSTRACT

A SIMULA class containing procedures for easily writing programs that interact with a user by asking questions at run time and which dynamically name and open files at run time is described. The class uses properties of IBM SIMULA that are not available in other implementations. It also depends on the EBCDIC character codes rather than ASCII but it is assumed that a user's terminal is an ASCII terminal.

This revised report describes a number of extensions of DIALOG. While most programs using the old version will work correctly with the new version, there are many enhancements and users are advised to reread all Sections 3 to 9.

Keywords and Phrases: SIMULA, interactive programming

CR Categories: 4.22, 4.49, 4.39

---

* SIMULA is a registered trademark of the Norwegion Computing Center, Oslo, Norway.

† Research sponsored by the Air Force Office of Scientific Research, Air Force Systems Command, under Grant No. AFOSR-70-0021. The United States Government is authorized to reproduce and distribute reprints for Govermental purposes notwithstanding any copyright notation hereon.

§ The information in this document is subject to change without notice. The author, Virginia Polytechnic Institute and State University, the Commonwealth of Virginia and the United States Government assume no responsibility for errors that may be present in this document or in the program described here.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFOSR-TR- 80-0446	2. GOVT ACCESSION NO. AD-A083 216	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) DIALOG - A SIMULA CLASS FOR WRITING INTERACTIVE PROGRAMS		5. TYPE OF REPORT & PERIOD COVERED Interim 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Richard J. Orgass Robert E. Porter		8. CONTRACT OR GRANT NUMBER(s) AFOSR 79-0021
9. PERFORMING ORGANIZATION NAME AND ADDRESS Virginia Polytechnic Inst. & State University Department of Computer Science Washington, DC 20041		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61102F 2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS Air Force Office of Scientific Research/NM Bolling AFB, Washington, DC 20332		12. REPORT DATE October 1979 13. NUMBER OF PAGES 26
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

SIMULA, interactive programming

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A SIMULA class containing procedures for easily writing programs that interact with a user by asking questions at run time and which dynamically name and open files at run time is described. The class uses properties of IBM SIMULA that are not available in other implementations. It also depends on the EBCDIC character codes rather than ASCII but it is assumed that a user's terminal is an ASCII terminal.

This revised report describes a number of extensions of DIALOG. While most

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. Abstract cont.

programs using the old version will work correctly with the new version, there are many enhancements and users are advised to reread all Sections 3 to 9.

1. Problem Statement

A continuing problem in the design of genuinely interactive programs is that it is quite painful to write code for reliably and meaningfully interacting with the user sitting at his terminal. This is true in all time sharing systems and is a particularly severe problem in the CMS system because the terminal interface provided by the operating system is quite crude.

The minimal sequence of events that must occur when a user is asked to provide input may be described as follows:

(1) Prompt the user with a question.

(2) Read the answer.

(3) Check the answer to make sure that it is an acceptable answer.

(4) If the answer is unacceptable, provide a corrective message and ask the question again.

There are two other properties of a question asking program that are particularly helpful. First, it is a good idea to provide a mechanism for a user to request an explanation of the question. If the user doesn't understand the question, he should be able to type "help" or "?" to request an explanation of the question in place of an answer. Second, it should be possible to provide a default answer that is the most typical answer. A user should be able to select this answer simply by hitting the "return" key.

Many programs perform useful work by processing files and writing other files. A very useful way of identifying these files is to read the file specification at run time. One would like to be able to prompt for a file name and then open the file to read or write and then request additional file names. This is, at best, difficult in CMS.

When processing text input, it is often useful to have a collection of procedures for testing strings to find out if they have specific properties or to extract substrings. A few simple procedures provide the capabilities that are needed in many applications but more elaborate applications are best served by using an implementation of the SNOBOL scanner. [A SIMULA coded coroutine implementation of the SNOBOL scanner is available from the first named author.]

This memorandum describes a SIMULA class that contains procedures for performing these operations. A SIMULA coded implementation of a coroutine SNOBOL scanner is available for applications that require extensive text processing capabilities.

Section 2 describes procedures for asking questions at a terminal and for examining a user's response and Sections 3, 4, 5 and 6 contain a description of file management procedures. Some useful utility procedures are described in Section 7 and Section 8 gives detailed information about gaining access to these procedures and incorporating them into SIMULA programs. There is a discussion of terminal attributes in Section 9 and the appendix contains an example of the use of DIALOG.

The terminal transcripts exhibited in this memorandum adhere to the following conventions. A running program prompts with an asterisk (*) when there is no other prompt and CMS at monitor level prompts with a period (.). All of the special characters (CHARDEL, LINEDEL, etc.) are nonprintable ASCII characters.

2. Query Procedures

All of the procedures described here have the following properties. When the procedure is called the parameters include the question to be asked, a default answer, and a procedure to print a further explanation of the question is printed on the terminal and the user's response is the return value. If the user responds with a carriage return, the return value is the default value and if the user responds with a question mark (?) the procedure to explain the question is invoked and then the question is asked again.

For example, suppose a program wishes to read an input file name from the terminal into a text variable file_spec. This can be accomplished by executing the following statement:

```
file_spec :- text_request("Input file:", NOTEXT, TRUE);
```

The first parameter is the question that is to be printed on the terminal. The second parameter is the default answer and since this parameter is NOTEXT, there is no default answer. The third parameter is the constant TRUE to indicate that no help is available. When this statement is executed, the terminal transcript will look like this:

```
Input file:?
No help available.
Input file:
? Default value may not be selected.  Please try again.
Input file:letter simula
```

After this, the return value of text_request is the text "letter simula". In the first response, the user asked for an explanation of the query by responding with a "?". Since the call to text_request did not include a help procedure, the appropriate message was printed. Next, the second response was an empty line indicating that the default value was desired. Since no default value was specified in the procedure call, a corrective response was printed and then the question was asked again. Finally, the third response was a character string and this string became the return value of the procedure.

As a second example, suppose that the statement:

```
f :- text_request("Output file:", "letter data, TRUE);
```

is executed. In this case the terminal transcript would look like this:

```
Output file:/letter data/: ?
No help available.
Output file:/letter data/:
```

If the second response was simply a carriage return, then the return value of text_request would be the string "letter data". On the other hand, if the response were some character string, then this string would be the return value.

Suppose it is desirable to provide a help message in response to the input "?". This might be accomplished by writing the following procedure:

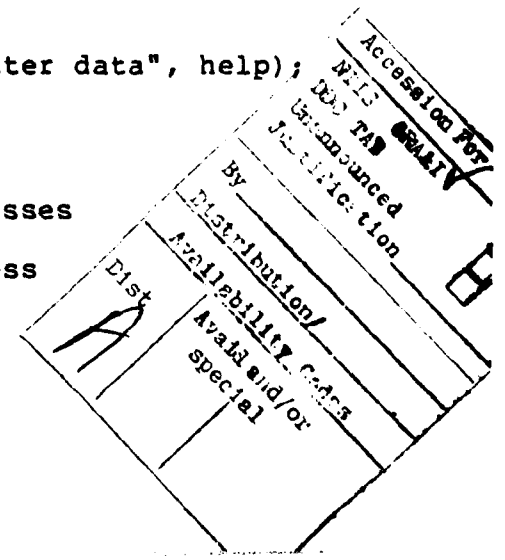
```
BOOLEAN PROCEDURE help;
BEGIN
    Outimage;
    Outtext("This file will contain ");
    Outtext("a list of addresses");
    Outimage;
    Outtext("after the program is executed.");
    Outimage
END of help;
```

The call

```
f :- text_request("Output file:", "letter data", help);
```

might generate the following transcript:

```
Output file:/letter data/: ?
This file will contain a list of addresses
after the program is executed.
Output file:/letter data/: mylib address
```



After this transcript, the return value of text_request is the string "mylib address".

The heading of the declaration of text_request is:

```
TEXT PROCEDURE text_request(prompt, default, no_help);
    NAME prompt, default, no_help;
    TEXT prompt, default;
    BOOLEAN, no_help;
```

The formal parameter prompt is the question to be printed on the terminal. The parameter default is the value to be returned if the user's response is a carriage return. The parameter no_help is to be TRUE if there is no help available for this query. If there is help available, it is printed by a boolean procedure that returns the value FALSE.

The procedure boolean_request is used to ask yes or no questions in very much the same way. The heading of its declaration is

```
BOOLEAN PROCEDURE boolean_request(prompt, default,
                                  no_help);
    NAME prompt, no_help;
    VALUE default;
    TEXT prompt;
    BOOLEAN default, no_help;
```

The parameter prompt is the question that is to be printed on the terminal and the parameter default (which must be TRUE or FALSE) is the return value if the user responds by entering "return". If the user enter responds with a "?" and if no_help is TRUE then the message "No help available." is printed. On the other hand, if no_help is a boolean procedure that returns FALSE and prints an explanation, this text is printed instead of "No help available."

For example, if the procedure help6 is declared as:

```
BOOLEAN PROCEDURE help6;
BEGIN
    Outtext("If tabs may be used at indentation ");
    Outtext("answer ""yes"",");
    Outimage;
    Outtext("otherwise answer ""no"".");
    Outimage
End of help6;
```

Then the execution of the statement

```
tabs := boolean_request("Tabs in indentation:", FALSE,
                        help6);
```

might result in the following transcript:

```
Tabs in indentation:/n/: ?
If tabs may be used at indentation answer "yes",
otherwise answer "no".
Tabs in indentation:/n/: why
Please answer y or n.
Tabs in indentation:/n/: y
```

After this transcript, the return value of boolean_request is TRUE. If the last response was an empty line or "n", then the return value would be false.

Note that the responses "y", "yes", "Y", "YEs" are all equivalent as are "n", "no", "N", "nO". More precisely, lower case letters are translated into upper case letters before the response is examined.

The procedure integer_request is used to prompt for an integer response and to check if the response is an integer. Further, if the response is an integer, it is checked for being in an acceptable range. The heading of this procedure is:

```
INTEGER PROCEDURE integer_request(prompt, default, min,
                                  max, no_help);
    NAME prompt, no_help;
    VALUE default, min, max;
    TEXT prompt;
    INTEGER default, max, min;
    BOOLEAN no_help;
```

The formal parameter prompt is the question to be printed on the terminal and the formal parameter default is the value that is returned if the user responds with a "return". After an integer is read from the terminal, it is checked to confirm that it is between min and max. If it fails this test, the user is asked for a correct answer. (The default value is also checked against the range if the user responds with "return"; this helps catch programming errors.) The formal parameter no_help is used to deal with the user response "?" as described above.

If any integer is an acceptable response, the SIMULA defined constant Maxint may be used in a call. For example, if the default answer to the prompt "Enter any integer:" is 0, one would execute the statement:

```
result := integer_request("Enter any integer:", 0,
                          -Maxint, Maxint, TRUE);
```

As a more detailed example, consider the execution of the statement

```
result := integer_request("reserved words:", 1,
                          0, 3, TRUE);
```

The terminal transcript might look like this:

```
Reserved words:/0/: ?
No help available.
Reserved words:/0/: bye
The input was not an integer.  Please try again.
Reserved words:/0/: 12
The input integer was out of the acceptable range [0,3].
Please try again.
Reserved words:/0/: 2
```

After this sequence of events, the return value of integer_request is 2.

These three procedures provide most of the terminal prompting activities that are needed. While it might be desirable to prompt for floating point numbers, the authors have not felt the need and, therefore, did not include it in this program.

An excerpt from a program that uses these procedures as well as a sample terminal transcript from the execution of the program appears in the Appendix.

3. Input File Management

Connecting files to a running program in CMS is, at best, a rather tedious process. CMS requires a substantial amount of information before the file can be opened and read. Three procedures that make it easier to read files are described in this section.

In SIMULA, a new Infile is created by executing the statement

```
f :- NEW Infile("DDN");
```

The parameter of Infile must be a text object whose value is the DD name of an input file. That is, before this statement is executed, a CMS filedef command such as

```
FILEDEF DDN DISK MUMBLE FOO
```

must be executed. If this command has not been executed, then a SIMULA run time system fatal error will occur. In addition, the filedef command must be executed before the program begins execution so it is impossible to read a file name during program execution.

After creating an Infile, it is necessary to open the file by executing a statement such as:

```
f.Open(Blanks(lrecl));
```

The value of the variable lrecl must be at least the record length of the input file. If the value of lrecl is less than the record length of the file, a SIMULA run time system fatal error will occur.

One would much prefer to be able to open a file by simply giving the CMS file specification at run time and leaving other details to the operating system or other programs. The procedures get_infile, get_dd_input and get_input_file make it possible to read files when only the CMS file specification is known. Moreover, the file names can be read during program execution.

The heading of the declaration of procedure get_infile is:

```
REF(Infile) PROCEDURE get_infile(t); NAME t; TEXT t;
```

The value of the parameter t must be a CMS file specification of the form <fn>[<ft>[<fm>]]. At least one space or a period must separate the three components of the file specification. If the <ft> is omitted, DATA is assumed as this component of the file specification. If <fm> is omitted, A is assumed. In addition, if there is no file with the given <fn> <ft> on the A disk all other disks are searched for files with this name and type. The first file in the (user defined) search order that has this <fn> <ft> is selected.

It is possible to omit the <ft> and select the default <ft> while specifying the <fm>. The string FOO..B refers to file FOO DATA B.

The return value of get_infile is an open Infile which is connected to the file named in the value of t. A filedef (using a standard DD name of the form SIMnnn) is issued during the call to get_infile and then the file is opened. The length of the Image attribute of this Infile is the record length of the file.

As an example, consider the following sequence of statements:

```
f :- get_infile("mumble foo");
lrecl := f.Image.Length;
f.Inimage;
t :- Copy(f.Image);
```

After the first statement is executed, an Infile connected to file MUMBLE FOO is created. When the second statement is executed, the value of lrecl becomes the record length of file MUMBLE FOO. When the third statement is executed, the first record of MUMBLE FOO is read into f.Image. The last statement assigns a copy of this text object as the value of t.

Notice that it is not necessary to know the record format of the input file! Either fixed or variable length records can be easily read using this procedure.

The procedure get_input_file makes it possible to easily prompt the user for a file name and then issue a filedef for the file. The heading of the declaration of get_input_file is:

```
TEXT PROCEDURE get_input_file(prompt, lrecl);
    NAME lrecl; VALUE prompt;
    TEXT prompt; INTEGER lrecl;
```

When this procedure is called, the first parameter is a text object whose value is included in a prompt to the terminal requesting an input file name. Upon return, the value of lrecl is the record length of the file whose name was read from the terminal. When this procedure is executed, the file specification is read from the terminal and then a filedef command for this file is issued (using a unique DD name of the form SIMnnn) and this text object is the return value of the procedure.

As an example of the use of this procedure, consider the following sequence of statements.

```
fname :- get_input_file("program", lrecl);
f :- NEW Infile(fname);
f.Open(Blanks(lrecl);
```

When the first statement is executed, the message

```
program input file ?
```

is printed on the terminal and then input is read from the terminal. This input is interpreted as a CMS file specification and a filedef for this file is issued. The DD name used in the filedef is the return value of the procedure and lrecl is set to the record length of the file. The next two statements create the Infile object and open the file. For example, if there is a file named MUMBLE FOO whose record length is 155 in the directory of the user executing this program, the terminal dialog might look like this:

```
program input file ? mumble foo
```

A filedef for this file would be issued and lrecl would be set to 155. The DD name assigned to the file will be the return value of get_input_file.

The heading of the declaration of get_dd_input is:

```
TEXT PROCEDURE get_dd_input(file_spec, lrecl);
    VALUE file_spec; NAME lrecl;
    TEXT file_spec; INTEGER lrecl;
```

The value of the first parameter is a CMS file specification using the syntax described above. When this procedure is executed, a filedef for this file is executed and the DD name

assigned to the file is the return value of the procedure. In addition, the value of lrecl is set to the record length of the file. This procedure is primarily used as a helping procedure for the other input file management procedures.

Error Checking

All of the procedures described here perform complete error checking on the file specification. If there is any error in the file specification, the user is provided with an error message and is asked to provide a corrected file specification. The abbreviations described above also apply to these corrective responses. If any of the three procedures described here returns a value, the action described above has been performed correctly! There will not be a SIMULA run time fatal error as a result of opening input files with these procedures.

When one of these procedures prints an error message concerning the file specification the default response (selected by entering <cr>) is the string "CMS:". When this response is read all subsequent input lines are executed as CMS commands and the CMS response is printed on the terminal. If a command outside the CMS subset is executed, the core image of the program will be destroyed; there is no error check for this condition! While in CMS mode, control is returned to the program by entering the string RETURN. After this, another prompt for the input file will be issued and a new response is requested.

This facility is quite useful when a user has forgotten the name of the input file. The CMS command list can be used to examine the directory to find the input file that is desired.

All of these procedures assign a special role to the string "tty:" (in upper, lower or mixed case). This string is the file specification for the terminal as an input device and is used to connect Infiles other than Sysin to the terminal.

4. Output File Management

In SIMULA, an output file is created by executing the statement

```
f :- NEW Outfile("DDN");
```

or

```
f :- NEW Printfile("DDN");
```

where DDN is a DD name as for Infile. After an Outfile is created in this way, the file is opened and the length of the

parameter of open defines the record length of the file to be written.

As for Infiles, a CMS filedef command must be issued for each DD name before program execution begins. Three procedures that simplify the creation of Outfiles and Printfiles are described in this section and an extension of class Outfile, class out_file is described in the next section.

The procedures get_outfile and get_printfile are the easiest way to create Outfiles and Printfiles and the procedure get_dd_output is primarily used as a helping procedure inside DIALOG.

```
REF(Outfile) PROCEDURE get_outfile(file_spec, lrecl);
    NAME file_spec;
    TEXT file_spec;
    INTEGER lrecl;
```

The first parameter of get_outfile has as its value a CMS file specification of the form

```
<fn>[ <ft>[ <fm>]]
```

and the second parameter specifies the record length with which this file is to be written. The return value of get_outfile is an open Outfile which will write output to the file named in the first parameter. The string "TTY:", when used as a file specification, indicates that output is to be written to the terminal.

When this procedure is executed, the following occurs:

(1) If the <ft> is omitted, it is set to "LOG".

(2) The file specification is checked for syntax errors and corrections are requested from the terminal if needed.

(3) A filedef for this file specification with a DD name of the form SIMxxx and LRECL set to lrecl is executed. (The RECFM is set to F.)

(4) A new Outfile is created using this DD name.

(5) This Outfile is opened with Image.Length=lrecl.

(6) The Outfile is the return value of the procedure.

For example, after the statement

```
f:- get_outfile("MUMBLE", 132);
```

is executed the value of f is an Outfile that will write to file MUMBLE LOG on the A disk. This file will have an LRECL of 132 and RECFM F.

```
REF(Printfile) PROCEDURE get_printfile(file_spec, lrecl);
    NAME file_spec;
    TEXT file_spec;
    INTEGER lrecl;
```

The first parameter of this procedure is a CMS file specification as described above and the second parameter is the LRECL of the file to be written. When this procedure is executed, the following occurs:

(1) If the <ft> of the file specification is omitted, it is set to "LOG".

(2) The file specification is checked for syntax errors and corrections are requested from the terminal if needed.

(3) A filedef for this file specification with a DD name of the form SIMxxx, with LRECL set to lrecl and RECFM set to "F" is issued. If there are errors executing the filedef, corrections are requested from the terminal.

(4) The DD name is the return value of the procedure.

These three procedures provide an adequate set of primitives for creating output files at run time but the class out_file described below is much easier to use.

5. Class out file

The system defined class Outfile suffers from a number of limitations which make it impossible to write some programs and awkward to write many other programs. Class out_file is designed to serve a replacement for class Outfile.

When the system defined class Outfile is used, it is impossible to transmit a line of output to the terminal without a trailing carriage return -- line feed pair. This means that responses to prompts must always be on the line after the prompt. Using the Breakoutimage attribute of class out_file, this kind of terminal dialog is easy to write.

When the system defined class Outfile is used, two calls to Outimage must precede a call to Inimage if the terminal transcript is to be in the right order. These extra calls to Out-

image are awkward to write and also introduce extra blank lines into the terminal transcript.

The parameter of class Outfile is a DD name rather than a file specification. This means that it is necessary to go through some procedure to convert a file specification into a DD name. The procedures described in Section 4 make this conversion very much easier but it is still an awkward way of writing programs.

In an interactive environment, many output errors can be corrected at run time by means of terminal prompts; these corrections are not provided in class Outfile.

Class out_file is the same as class Outfile except for the following changes:

(1) The parameter of class out_file is a CMS file specification of the form

    <fn>[<ft>[<fm>]]

    or the string "TTY": (in upper, lower or mixed case). This file specification determines the destination of output. If <ft> is omitted, the string "LOG" is provided. The device name "TTY:" refers to the terminal. This parameter is also a read only attribute, file_spec, of the class.

(2) Class out_file has a niladic procedure attribute Breakoutimage with the following property. When this procedure is executed from an instance of out_file whose file_spec attribute is "TTY:", the string

    Image.Sub(1, Image.Pos)

    is written to the terminal without a trailing carriage return -- line feed. If the file_spec attribute is not the string "TTY:", then the usual call to Outimage is executed.

(3) When the file_spec attribute of an instance of out_file is "TTY:", only one call to Outimage is required to keep the terminal transcript in order. For other file specifications, the usual call to Outimage is executed.

(4) In the system defined class Outfile, if a call to Outtext is executed with insuffi-

cient space for the text in Image, an error termination occurs. In class out_file, this text is written on as many lines as are needed to print the text. This avoids many unexpected error terminations.

(5) In the system defined class Outfile, if there is an attempt to transfer data via a closed instance of the class, an error termination results. In class out_file, the user is given an opportunity to open the file and, thus, avoid error termination. Error terminations are, of course, still an acceptable action but the user makes the decision at run time.

(6) Procedure attributes for writing ASCII text to keyboard and bit paired ASCII/APL terminals are provided (See Section 6).

The text of a program that uses class out_file should be structured as follows:

```
BEGIN
    EXTERNAL CLASS dialog;
    dialog BEGIN
        INSPECT tty DO
        BEGIN
            <main block of program>
        END of tty block;
        restore_terminal
    END of dialog block;
END of program.
```

When this is done, an instance of out_file (the value of tty) is the default output device instead of Sysout. The appropriate initializations are performed in dialog.

6. APL Compatability

In some applications, a program will be reading from and writing to a key or bit paired ASCII/APL terminal and perform its own character set translation. These programs might well be using class DIALOG or find other reasons to write ASCII text to an output device. If this is done without translation, the messages are unintelligible. The procedures Outimage and Breakoutimage in class out_file provide for character translation in accord with the terminal type. [For normal ASCII output the overhead associated with this translation is negligible (one comparison) and later versions of out_file will be far more efficient than the system defined version.]

Users who are not working with ASCII/APL terminals need not read this section -- all defaults in DIALOG assume an ASCII terminal.

Class out_file has six procedure attributes related to APL character sets. Four of these procedures are used to control the output character set and two are used to perform output operations.

The procedure term_type returns an integer that is related to the output character set as follows:

Character Set	term type
ASCII	0
key_paired APL	1
bit_paired APL	2

These integer values were chosen to match the values of the system variable quad-TT in APL implementations.

The STAPL convention for ASCII/APL terminals specifes that the character SO (ASCII 14, control-N) causes a terminal to switch to the APL character set and that the character SI (ASCII 15, control-O) causes a terminal to switch to the ASCII character set. These conventions are followed in class out_file.

When the procedure set_ascii is executed, the value of term_type becomes 0 and no output character translation is performed after the procedure is executed. In addition, a line containing the character SO is written to the output device. This causes the execution of Outimage to empty Image before the character is written; the previous translation is in effect for this line.

When the procedure set_key_paired is executed the value of term_type becomes 1 and after this call all output via APL_outimage and APL_breakoutimage is translated for key_paired ASCII/APL terminals as described below. In addition, a line containing the character SI is written to the output device. This causes the execution of Outimage to empty Image before the character is written; the previous translation is in effect for this line.

The procedure set_bit_paired is the same as set_key_paired except that term_type is set to 2 and translation is performed for bit_paired ASCII/APL terminals.

ASCII text is translatd into APL subject to the following convention. Lower case ASCII letters are translated into APL letters and upper case ASCII letters are translated into underlined APL letters. The ASCII graphics are translated into the corresponding APL graphic. The following ASCII graphics are translated

into different APL characters because there is no exact equivalent in the APL character set.

ASCII Graphic	Translated Graphic
#	right tack
%	diamond
&	and sign
@	alpha
^	cent sign
`	high minus

In a very limited number of applications it may be essential to bypass automatic character translation. For example, a program that directly writes to a terminal or file in the APL character set might wish to bypass translation. The procedure apl_outimage and apl_breakoutimage are just like Outimage and Breakoutimage except that no translation is performed independent of the terminal type.

7. Utility Procedures

A number of procedures that make it easier to write interactive applications are described in this section. Some of the procedures were written specifically for the IBM SIMULA environment; some are adaptations of procedures that are in the DEC-10 SIMULA library and others are taken directly frm the DEC-10 SIMULA manual.

The utility procedures are described by exhibiting the heading of their declaration and following this with a brief description of the procedure.

```
TEXT PROCEDURE frontstrip(t); TEXT t;
```

This procedure returns a subtext of t that has all leading blanks removed. The value of the expression

```
frontstrip(x.strip)
```

is the text x with both leading and trailing blanks removed.

```
TEXT PROCEDURE upcase(t); TEXT t;
```

The return value of this procedure is a new text object that is the same as its actual parameter except that all lower case letters are changed to the corresponding upper case letters. It does not map lower case national letters in the ISO standard into upper case national letters.

```
TEXT PROCEDURE rest(t); TEXT t;
```

The return value of this procedure is a subtext of t that begins at t.Pos and ends at t.Length.

```
BOOLEAN PROCEDURE is_integer(t); NAME t; TEXT t;
```

This procedure returns the value TRUE if the text object t is an integer and the value FALSE otherwise. A text object is an integer if the first character is '+', '-' or a digit and if the remaining characters in t are digits.

```
TEXT PROCEDURE conc2(t1, t2); VALUE t1, t2;
    TEXT t1, t2;
```

The return value of this procedure is a new text object that is the concatenation of its two text parameters in the order first parameter, second parameter.

```
PROCEDURE next_file(f); REF(Infile) f;
```

This procedure closes file f and then opens it again with the same record length as it had before the close. This procedure is useful when writing programs that accept an empty line as selecting the default answer. In CMS, this empty line is treated as an end-of-file and an attempt to read another record results in an error termination. By executing next_file after the read, the error termination can be bypassed. Here is an example of code to do this:

```
IF Sysin.Image.sub(1,2) = "/*"
    THEN next_file(sysin);
```

```
PROCEDURE tty_inimage(f); REF(Infile) f;y
```

This procedure is the same as the system defined Inimage attribute of Infiles with one exception. When an end-of-file is encountered two actions are taken:

(1) The value of f.Image is set to Blanks(f.Image.Length).

(2) File f is closed and then opened again with the same record length.

This procedure is designed for constructing interactive programs in such a way that an empty line entered from a terminal can be treated as an empty line rather than as an end-of-file (the CMS convention).

```
PROCEDURE cms_subset;
```

When the procedure cms_subset is executed, the following happens:

(1) The terminal prompt character is sharp (#).

(2) Each input line typed by the user is transmitted to CMS for execution. After the command is executed, CMS output is typed on the terminal and the usual ready message is printed.

(3) When the input line return is encountered, control is returned to the calling program.

```
TEXT PROCEDURE insert_tabs(t);
    VALUE t;
    TEXT t;
```

The return value of this procedure is a copy of the text object t.Strip with blanks replaced by tabs (ASCII 9, EBCDIC 5) under the assumption that tabs are set every eight spaces as per the ANSI standard. Many text objects will require substantially less space after processing by insert_tabs. The SIMULA compiler and the spooling utilities described in the CMS Software Notebook (TM 79-6) will correctly process files with imbedded tabs.

```
TEXT PROCEDURE expand_tabs(t);
    VALUE t;
    TEXT t;
```

The return value of this procedure is a copy of the text object t with tabs (ASCII 9, EBCDIC 5) replaced by the appropriate number of spaces under the assumption that tabs are set every eight spaces as per the ANSI standard.

8. Directions

All of the procedures described above are included in a class DIALOG. The design of this class was motivated by the implementation of a class SAFEIO by Mats Ohlin of the Swedish Research Institute of National Defense in DEC-10 SIMULA. The present design was tailored to meet the needs of IBM SIMULA users and, therefore, differs in many details from SAFEIO.

To incorporate these procedures in a program, the program structure should be as follows:

```
BEGIN
    EXTERNAL CLASS dialog:
    dialog BEGIN
        INSPECT tty DO
        BEGIN
            <text of program using dialog>
        END of tty block;
        restore_terminal
    END of dialog block;
END of program.
```

If this program is contained in a file named TEXT SIMULA, it is compiled with the CMS command

```
SIMULA TEXT (CLASS DIALOG <other options>
```

Before compiling a program that uses DIALOG, you should either copy the SIMCLASS file onto your disk or link to the Computer Science library disk. To copy the SIMCLASS file onto your disk, execute the following commands:

```
LINK CSDULLES 191 333 READ ALL
ACCESS 333 G
COPY DIALOG SIMCLASS G DIALOG SIMCLASS A
DETACH 333
```

If you prefer to use the library copy of these files, simply execute the commands:

```
LINK csdulles 191 333 read all
ACCESS 333 G/A
```

Using the library copy has the advantage that you will be using the most current copy of DIALOG.

9. Terminal Attributes

The input/output procedures make certain assumptions about properties of the CMS environment in which the program that uses these procedures is executed. As part of the initialization of DIALOG, attributes of the environment are set and it is possible to restore the environment when a program completes execution.

It is assumed that the user of an interactive program does not want to receive messages while the program is in execution and, therefore, WNG, MSG and ACNT are set OFF during DIALOG initialization. These attributds are turned on again when the procedure restore_terminal is executed.

It is assumed that interactive programs will generate their own prompts either by using the query procedures (Section 2) or

directly using Breakoutimage and, therefore, terminal prompting is turned off during DIALOG initialization and the prompt character is restored to period (.) when restore_terminal is executed.

It is assumed that terminals are constructed so that the first character on line i+1 immediately follows the last character on line i. This means that it is possible to write several lines of output with a single write. In order to take advantage of this property, the terminal line length is set to the CMS limit of 255 characters; this is als tty.Image.Length.

It is assumed that interactive programs will read upper and lower case letters as different characters and, therefore, the appropriate FILEDEF for SYSIN is executed.

Since many of the usual LINEND characters are used in some application, LINEND is turned off during program execution and set to escape or altmode (ASCII 27, EBCDIC 39) when restore_terminal is executed.

The initializations performed by DIALOG are such that modules created from SIMULA programs can be executed directly without any supporting JCL.

In summary, during DIALOG intialization, and whenever the procedure initialize_terminal is executed, the following occurs:

(1) CP TERM PROMPT OFF
(2) CP TERM LINEND OFF
(3) CP TERM LINES 255
(4) CP SET WNG OFF
(5) CP SET ACNT OFF
(6) CP SET MSG OFF
(7) Sysin is opened as an upper and lower case file.
(8) tty is opened as an instance of out_file with a record length of 255.

When the procedure restore_terminal is executed, the following occurs:

(1) CP TERM PROMPT .
(2) CP TERM LINEND <esc>
(3) CP SET WNG ON
(4) CP SET MSG ON
(5) CP SET ACNT ON

APPENDIX

EXAMPLE OF USE OF DIALOG

This appendix contains a fragment of a program that edits the text of SIMULA programs. This program can completely reformat the text of an input file and change all of the properties of the text that influence its appearance. In addition, it is capable of inserting tab characters to reduce the disk space required to store the text.

The program interacts with the user by asking a long sequence of questions that define the behavior of the program. Each of these questions has a default answer and it is possible to select the defalt answer by terminating the first response with the character escape or altmode.

The following paragraphs describe the statements in the program. The reader is encouraged to examine the program text that follows the explanation as the text provides a clear example of the use of DIALOG procedures. A sample terminal transcript follows the program text.

The variable fastflag is set to TRUE if the user indicates that he wishes to select the default answers to all questions. At the beginning of the dialog, this variable is set to FALSE.

The first question asks the user to provide the name of the file that contains the program. This is done with a call to text_request. Since there is no default file name, the second parameter of text_request is NOTEXT. There is a help procedure for this query called help1 elsewhere in the program.

The specifications of this program state that if the input file name is terminated with the character escape or altmode then the default answer to all of the remaining questions are selected. After the file name is read, leading blanks are removed with a call to the procedure frontstrip. Next the last character of the user's response is examined to find out if it is escape (ASCII 27, EBCDIC 39).

If this character is present, fastflag is set to TRUE and an instance of Outfile for the output is created. The specifications indicate that the default file specification of the output file has the same file name as the input file and file type SIMED and that the record length of this file is to be 80. In the next statements, the escape character is removed from the file specification and then the variable outf is set to the appropriate instance of Outfile using the procedure get_outfile.

The specifications for this program also state that if the file type of the input file specification is omitted, then the file type of the input file will be set to SIMULA. The complete file name is composed in the next statement using the procedure conc2 to compose the complete file specification and get_infile is used to set the variable prog to the instance of Infile that will read the input file.

At this point, the input file has been initialized and if fastflag is true, the output file has also been initialized. If fastflag is FALSE, it is necessary to read the name of the output file from the terminal. The specifications of this program also state that if the name of the output file is terminated by the character escape then the default answers to the remaining questions will be assumed. This processing as well as assigning the variable outf the appropriate value is done in the IF statement that begins IF NOT fastflag.

These are the only two questions that permit the use of the escape character to select the default answers to the remaining questions. Therefore, the next statement sets these default values if fastflag is true and then skips the remaining questions.

The next question asks the user to specify the number of characters in each indentation step. A negative answer means that leading blanks in the program text will be retained and, therefore, the allowable range of this answer is from a negative number to a positive number.

The next question asks the user to provide the rightmost position on a line where an indented line of text may begin. The smallest possible value is indent and the largest possible value is outlength.

A user is permitted to ask to have tabs used when writing the output file. The next question, using boolean_request, asks for directions. Since the SIMULA compiler accepts input files that contain tabs and output files are intended for processing by this compiler, the default answer is "yes".

The remaining questions ask the user to select conversion modes for different syntactical objects in a SIMULA program. A correspondence between integers and the conversion modes is printed on the terminal. After these modes are described, the user is asked to specify a conversion mode for reserved words, standard identifiers, user identifiers, comments and options and, lastly, for text constants. The appropriate program variables are set to the response to these questions.

The label fast appears at the end of these questions. The remainder of the program text consists of the code to perform the conversion.

The program text follows.

```
fastflag := FALSE;
progname :- text_request("Enter program file name:",
                         NOTEXT,
                         help1);
progname :- frontstrip(progname);
IF progname.Sub(progname.Length,1).Getchar = Char(39)
   THEN BEGIN
            fastflag := TRUE;
            progname :- progname.Sub(1,progname.Length-1);
            progname :- progname.Strip;
            outf :- get_outfile(conc2(first_token(progname),
                                      " SIMED"), 80);
        END;

prog :- IF no_blanks(progname)
           THEN get_infile(conc2(progname, " SIMULA"))
           ELSE get_infile(progname);

IF NOT fastflag
   THEN BEGIN
            outname :- text_request("Enter output file name:",
                                    conc2(first_token(progname),
                                          "_SIMED"),
                                     help2);
            outname :- frontstrip(outname);
            IF outname.Sub(outname.Length,1).Getchar
                        = Char(39)
                THEN BEGIN
                        fastflag := TRUE;
                        outname :-
                          outname.Sub(1,
                              outname.Length-1).Strip
                     END;
            outf :- get_outfile(outname, 80);
        END;
IF fastflag
   THEN BEGIN
            outlength := 72;
            indent := 0;
            leftskip := FALSE;
            maxindent := 52;
            tabs := FALSE;
            convert(2) := 1;
            convert(3) := 3;
            convert(4) := 2;
            convert(5) := 0;
            convert(6) := 0;
            GO TO fast
        END;
indent := integer_request("Enter indentation step:",
                0, (-outlength//2), outlength//2,help4);
```

```
leftskip:= indent > 0;    indent:= Abs(indent);

maxindent := integer_request("Enter max. indentation position:",
                52, indent, outlength,help5);
IF maxindent < 1 THEN maxindent:= 1;
tabs := boolean_request("Tabs in indentation?:",
                TRUE, help6);

Outtext("Conversion modes:");   Outimage;
Outtext("No change                  0");  Outimage;
Outtext("Change to upper case       1");   Outimage;
Outtext("Change to lower case       2");  Outimage;
Outtext("Change to edit case        3");
Outimage; Outimage;
Outtext("Enter conversion modes for:");   Outimage;
convert(2) := integer_request("Reserved words:",
                1,0,3,TRUE);
convert(3) := integer_request("Standard identifiers:",
                3,0,3,TRUE);
convert(4) := integer_request("User identifiers:",
                2,0,3,TRUE);
convert(5) := integer_request("Comment and options:",
                0,0,2,TRUE);
convert(6) := integer_request("Text constants:",
                        0,0,2,TRUE);
fast:
```

A terminal transcript of the execution of this program follows. The first line is a CMS command to load and execute the program SIMED. The message "EXECUTION BEGINS..." is emitted by CMS when the loader finishes its work.

The next line of output introduces the program. Instructions to print this message precede the text considered above.

After this, the prompt for the program file name appears and the user answered "dialog2". Since the name was not followed by escape, the next question concerned the output file name. The default name "dialog2 SIMED" is provided. The user selected this answer by entering "return".

Next, the indentation step was requested. The user found the default value unacceptable and provided 4 as his response. A value other than the default value was also selected for the question concerning the maximum indentation position.

The user accepted the default of tabs in the output file and then the program printed the conversion modes.

The user's responses to the questions concerning conversion modes appear in the transcript.

After all the questions were answered, the program text was converted and the program printed a message summarizing its work. The next line is the CMS ready message and the period on the next line indicates that CMS is expecting another command.

The help procedures were not exhibited in this example because the text is of little interest here. The code of these help procedures is very similar to the example given in Section 2.

```
.go simed
EXECUTION BEGINS...

SIMED - SIMULA EDITOR AND INDENTATION PROGRAM. IBM VERSION 1.0.

Enter program file name: dialog2
Enter output file name:/dialog2 SIMED/:
Enter indentation step:/0/: 4
Enter max. indentation position:/52/: 48
Tabs in indentation?:/y/:
Conversion modes:
No change                   0
Change to upper case        1
Change to lower case        2
Change to edit case         3

Enter conversion modes for:
Reserved words:/1/: 1
Standard identifiers:/3/: 1
User identifiers:/2/: 1
Comment and options:/0/:
Text constants:/0/:
[SIMED - Number of BEGIN's (END's) found:    43 ]
R;

.
```

END

DATE
FILMED

7-80

DTIC